Watching the File System
Reporting Acts of Creation
© 1999 by Andrew S. Downs
andrew@downs.net

Abstract
It is often useful to know when file-related actions occur. Typically, this requires either patching the File Manager or periodically searching one or more folders for items of interest. You then need to manipulate the resulting information and do something useful with it. This paper describes the design and implementation of a program which patches, processes, and presents information regarding file creation.

Introduction
In this paper, I describe the components of a system which can be used to obtain and relay information about file-related events to users. The term "user" will be used to refer to both end users and developers, unless specified otherwise.

Why do it?
Sometimes you want or need to know what is happening in the file system. However, the Macintosh file system (HFS and HFS+) does not contain ways to easily discover file-related events. (Folder actions in Mac OS 8.5 help somewhat.) You may wish to know when a file of a certain type is created, or when files in a particular folder get deleted, or maybe track what folder a user saves to most often (perhaps for the purpose of providing a helper application). These types of actions can form the basis of new products, or enhance an existing product. But to get that information, you need to interact with the file system.

How to do it?
The approach outlined in this paper consists of breaking down the information retrieval and notification process into manageable subprocesses, each of which can be addressed with a specific piece of code.

Overall, we want to:
- obtain information about file-related events
- massage the file information if necessary
- relay the information to a user

Each of these activities or subprocesses can be handled with code. But, how best to organize such code?

Different components
Let's consider the first goal: obtaining information about file-related events. This is exactly what patches (in the form of system extensions) are all about. In our case, we do not want to manipulate results of any traps, but we do need to know whether an operation succeeded. This implies the following characteristics of the patch:
- we need the parameters prior to execution (a head patch)
- we need the result code after execution (a tail patch)

In order to massage the file information, we need an intermediary between the patch and the user. This piece of code interacts with the patch in order to obtain whatever information the patch has been able to discover. This requires that the patch and intermediary define a structure (or class) that can be used to pass file information.

Another, possibly stronger, reason for an intermediary, is to reduce the amount of work performed by the patch. This is especially critical for traps that are called often, or that might be executing 68k code. Performance suffers if the patch has to do a lot of work before returning. Providing an intermediary solves some of the performance problem.

The approach outlined in this paper uses a background application (typically of type `'appe'`) as an intermediary. Background apps

should be unobtrusive and reliable, while still getting enough CPU time to do their work. Implementing and debugging background apps can be tricky; refer to Apple's Technote 1070 for details.

For exchanging data between the patch and background app, the File System Specification (FSSpec) record is nearly ideal. We will see that there are some additional fields that may be of interest. Wrapping the FSSpec in a new structure is one solution. Another is to use a Parameter Block Record (PBRec), which is how the low-level File Manager traps pass data. One issue with PBRecs is that they are unions of other record types, and so contain many additional fields that will not be of interest, resulting in wasted space.

A third component of our system is an interface with the user / developer. This can take one of several forms, depending on need. Four possible solutions are discussed later in this paper. Two of these (aliases and log files) are targeted at end users, while the others (Apple Events and callbacks) are intended for developer use.

Finally, there should be some way of controlling the action of the other components. A control panel is very useful for this purpose. The references listed in the bibliography provide a lot of detail on how to write control panels. Consequently, I will not address their implementation in this paper.

Figure 1 illustrates the various components of our system, and how they interact.

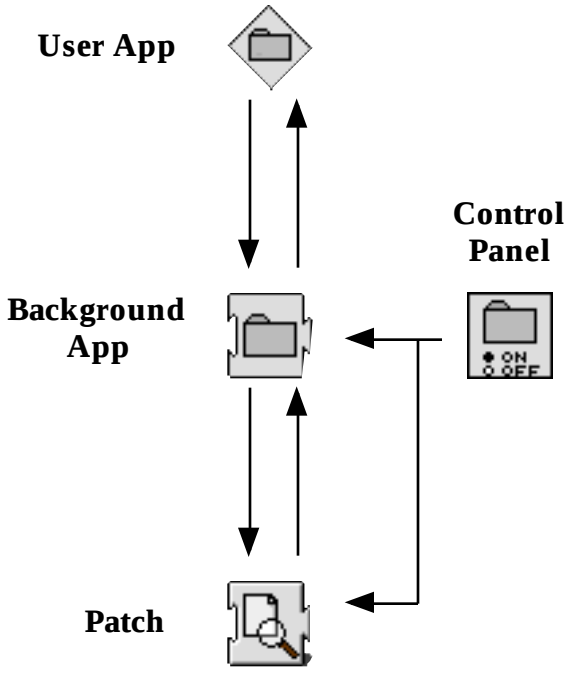

Figure 1. Components of our file system watcher.

Data flow

The patch obtains initial information about the file event by intercepting a File Manager call. It also gets the result code back from the File Manager upon completion of the original call. The patch hands the file information to the background app on request (when polled) or asynchronously (a push to a shared data area). Both methods work; each has its advantages.

The background app gets the raw data from the patch and performs additional file system queries in order to clarify or provide additional detail regarding the call. For instance, FSSpec information is not useful to the average user, but the path to the file is. Calling the File Manager to get the full path to a file specified in an FSSpec may be one of the background app's roles.

The interface to the user provides:

- useful information about file-related events (e.g. paths, timestamps)
- a way for the user to user to specify items of interest
- a way for the user to control patch and background app operations

The relay of information to the user can be accomplished through one of these methods:
• create an alias
• log to a file
• relay data using an Apple Event
• relay data using a callback

Other approaches can be devised, but only the four methods listed above will be discussed.

Ease of updating
Separating the components by function or purpose makes it relatively easy to update each piece as needed, and post those (smaller) updates to public archives or directly to registered users. A more compelling reason is that it is much easier as a developer to track changes in your code, and isolate testing, if you can focus on one or several small components rather than one monolothic application.

Items of interest
What, specifically, should our system be prepared to handle? Three possible requirements are what, where, and when an activity occured.

What happened
We need to get information about one or more of the following activities for files and / or folders:
• create
• copy
• delete
• rename / move

Each of these can be implemented using a patch to one or more File Manager traps, either high- or low-level. We will look at the file create activity, but the same approach can be applied to the other three.

Where it happened
As stated earlier, FSSpecs are useful in further querying the File Manager, but paths are often more useful to the end user. Depending on what patches we've applied, and what we intend to tell the user, we need:
• path to source
• path to destination

This applies to both files and folders.

When it happened
If we are simply providing an alias to a file, the operating system will ensure that the correct creation / modification date gets associated with the alias. But if we are writing to a log file, we also need:
• timestamp of the activity

The timestamp can be obtained by querying the File Manager or generating our own. The File Manager approach is more accurate, given potential system and network latency.

The patches
The patches discussed in this section have been applied in several products. The techniques shown are a combination of several presented in the references listed at the end of this paper.

Determine what to patch
For each "what" item, we need to determine the possible traps to patch. An API reference is indispensible for this purpose. Inside Macintosh and Think Reference contain cross-references and descriptions which help immensely. For instance, we can look up `Create()`, `HCreate()`, and `DirCreate()` to determine which of these traps suit our needs for file and folder creation. We also need to consider `PBCreate()` and `PBHCreate()` in their synchronous and asynchronous forms.

Purpose of the patches
As stated previously, the patches are as benign as possible (while still being useful). The intent is to gather, not change, information being passed to or received from the original trap.

How to patch
The patch code can be very simple. Recording the desired information is easier on the PPC, since you can avoid assembly language completely. However, the 68k assembly required is straightforward.

All of the patches listed below behave in the same manner as the patch code presented in Listing 1:

- `PBHCreate()`
- `PBHDelete()`
- `PBHRename()`

Listing 1 contains code to patch `PBHCreateSync()`. The `ifndef` handles 68k and PPC compiles. Both versions of the function follow a similar sequence.

The source code for the 68k macros is provided on the conference CD, along with the contents of the code listings. In the listings presented in this paper, some code has been removed to save space.

```
#ifndef powerc
asm  void  MyPBHCreateSync(
           HParmBlkPtr paramBlock ) {

  MacroPreProcess

  move.w  kPBHCreateSync, -( sp );

  MacroSaveValues

  move.l  oldPBHCreateSyncAddress, a1;

  MacroCallOriginalTrap
}
#else
OSErr  MyPBHCreateSync(
       HParmBlkPtr paramBlock ) {

  OSErr  theResult;

  //  Hold the params and the trap id
  //  temporarily.
  CopyHParamBlockValues( paramBlock,
    kPBHCreateSync );

  // Call the original trap.
  theResult = CallOSTrapUniversalProc(
    ( UniversalProcPtr )
      oldPBHCreateSyncAddress,
    uppPBFileProcInfo, paramBlock );

  //  Check result. On success,
  //  save the data.
  if ( theResult == noErr )
    SetArrayElement();

  return theResult;
}
#endif
```

Listing 1. Patching PBHCreateSync().

Now what?

Once the patch receives information regarding the file event (in the form of a PBRec), it needs to store that info until the original trap returns. If the call was successful, the retained data should be enqueued for the background app to process (in the push model), or stored in an array until requested (in the polling model).

Background processing

Now the background app must retrieve / receive data from the patch. Figures 2a and 2b illustrate the interaction using two approaches.

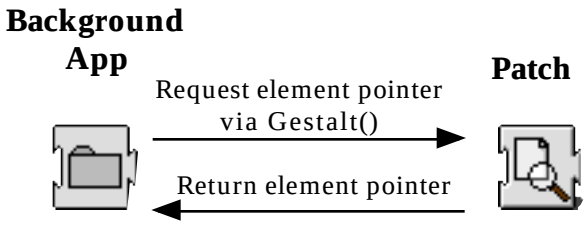

Figure 2a. Interaction using Gestalt().

Figure 2a illustrates a polling model. Here, the background app requests the address of the patch's Gestalt handler, which the patch advertises via `NewGestalt()`. Then, the background app uses a selector value to request the next element from the patch.

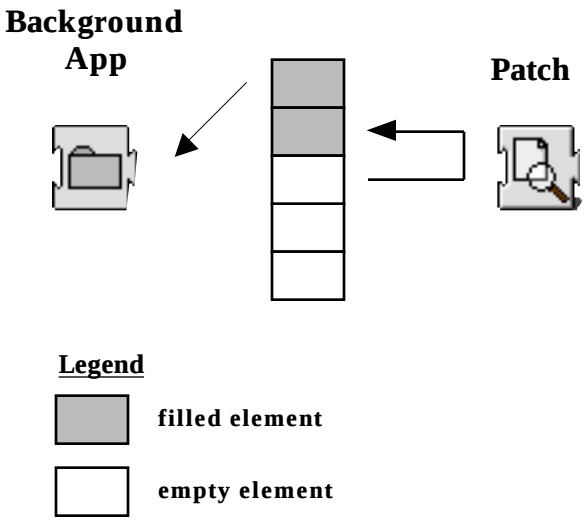

Figure 2b. Interaction using shared memory.

In Figure 2b, the background app and patch share a set of data structures. The patch retrieves the next empty element, fills it, and returns it to the pool. The background app retrieves the next filled element. There is no need for direct communication, except to setup the shared memory area. Determining whether an element is empty or not can be done by checking its values. The background app needs to clear the element contents after use.

Requesting data from a patch

In the polling model, `Gestalt()` can be used to obtain a pointer to the patch's routine through which it will provide the data. Then, calling that routine with an appropriate selector will result in the return of a struct containing the file info. An example of how to do this is presented in [Mark94]. A partial sample is presented in the next few listings.

Here is a sample data structure that can be used to hold the FSSpec data, plus a little more (such as the number of the trap that was called).

```
typedef  struct  FSPatchGlobals {
  short          theTrapId;
  short          theVRefNum;
  long           theParID;
  Str63          theString;
  OSType         theFileType;
} FSPatchGlobals, *FSPatchGlobalsPtr;
```

For this example, we will use an array of structs to track the calls to _Create.

```
FSPatchGlobals  gFSPatchGlobals[
                  kMaxNumFiles ];
```

Here is part of a Gestalt handler that can be used in the patch to respond to the background app's request for the next element.

```
static  pascal  OSErr  FSPatchGestalt(
                OSType theSelector,
                long *theResponse ) {

  switch ( theSelector ) {

    case kGetNextElement:
      //  Return the next element.
      *theResponse = ( long )
        &gFSPatchGlobals[
          gLastRetrievedElement ];

      //  Update the index.
      gLastRetrievedElement++;

      // Wrap to zero.
      if ( gLastRetrievedElement >
           kMaxNumFiles - 1 )
        gLastRetrievedElement = 0;

      break;
  }
}
```

Listing 2. Using Gestalt() to return data.

The background app can find and call the patch Gestalt handler. First, find the handler address:

```
OSErr      theErr;
static     pascal  OSErr
           ( *FSPatchGestalt )
           ( OSType selector,
             long *response );

theErr = Gestalt( kPatchSig,
  ( long * )&FSPatchGestalt );
```

Send the appropriate selector to request the next array element:

```
theErr = FSPatchGestalt(
```

```
kGetNextElement, ( long * )
&gRAGlobalsPtr[ gLastProcessed ] );

gLastProcessed++;
```

Listing 3. Using Gestalt() to request data.

Receiving data from a patch

Using a shared data area, the patch must first populate a structure or object, as shown in Listing 4. (An `XFileWatcherRecord` is similar, though not identical, to the `FSPatchGlobals` structure discussed previously.) The `HParmBlkPtr` is the source for info for the newly created file.

```
HParmBlkPtr pb;

XFileWatcherRecord *rec =
  (XFileWatcherRecord*)
  PopAtomicStack( gInputStack );

if( rec ) {
  rec->rec.action = kCreatedFile;
  rec->rec.vRefNum =
    pb->fileParam.ioVRefNum;
  BlockMovePString(
    pb->fileParam.ioNamePtr,
    rec->rec.name1 );
  rec->rec.dirID1 =
    pb->fileParam.ioDirID;

  PushAtomicQueue(
    (AtomicElement*) rec,
    gOutputQueue );
}
```

Listing 4. Using a shared data area.

Now the background app must retrieve the next available (filled) structure or object from the shared data area. It can use, and then clear, the object before returning.

```
XFileWatcherRecord *rec =
  (XFileWatcherRecord*)
  PopAtomicQueue( gOutputQueue );
```

If you are not familiar with atomic queues, you should definitely read “Atomicity” [Rentzsch99] while you’re here at the conference.

Determining something useful

After receiving the raw data for a newly created file, the background app can obtain the path to that file using `PBGetCatInfo()`.

If necessary, we can add our own timestamp using `GetDateTime()`, or query the OS for the creation time of the file using `PBGetFInfo()`.

Those Toolbox calls are well-documented in Inside Macintosh, THINK Reference, etc. Plus, relevant examples abound in Macintosh programming books. Some relevant code fragments are provided in the next section.

Informing the user

There are four methods of providing info to the user or developer. Of these, aliases and log files are suitable for end users. Callbacks and Apple Events are intended for developers.

Creating an alias

Creating an alias is a little involved. I like the example in [Little91]. Below are a few code snippets that handle potential problems.

Remember to put the alias file somewhere meaningful. You may want to create your own folder to hold the aliases. The following code checks if a named folder exists, and if not, creates it.

First, find the parent folder of our intended destination folder. Here, the parent is Apple Menu Items, so our folder should appear in the Apple Menu.

```
long    theFolderId;
CInfoPBPtr  thePBPtr;
FSSpec  theFSSpec;
OSErr   theError;

theError = FindFolder( kOnSystemDisk,
  kAppleMenuFolderType,
  kCreateFolder, &theFSSpec.vRefNum,
  &theFSSpec.parID );
```

Create the new directory. If it already exists, the call will fail, and we need to get the existing dir id using `PBGetCatInfo()`.

```
theError = DirCreate(
  theFSSpec.vRefNum, theFSSpec.parID,
  "\pMy Folder", &theFolderId );

if ( theError != noErr ) {
  theError = DoFindOneFolder(

  fileError = FindFolder(
    kOnSystemDisk,
    kAppleMenuFolderType,
    kCreateFolder, &theFSSpec.vRefNum,
    &theFSSpec.parID );

  thePBPtr->dirInfo.ioCompletion = 0L;
  thePBPtr->dirInfo.ioNamePtr =
    "\pMy Folder";
  thePBPtr->dirInfo.ioVRefNum =
    theFSSpec.vRefNum;
  thePBPtr->dirInfo.ioDrDirID =
    theFSSpec.parID;
  thePBPtr->dirInfo.ioFDirIndex = 0;

  theError = PBGetCatInfo( thePBPtr,
              true );

  while ( thePBPtr->dirInfo.ioResult ==
          1 ) { ; }
```

Check that the call succeeded, and returned a valid directory id.

```
  if ( ( thePBPtr->dirInfo.ioResult ==
         noErr ) &&
       ( theError == noErr ) &&
       ( thePBPtr->dirInfo.ioFlAttrib &
         0x0010 ) )
    theFolderID =
      thePBPtr->dirInfo.ioDrDirID;
}
```

Listing 5. Locating the destination directory.

Now use that directory id when creating the alias file. The last line of code in this listing establishes our special folder as the destination for the alias.

```
FSSpecPtr  aliasSpec;
OSErr   fileError;

BlockMove( theFSSpec.name,
           aliasSpec.name,
           theFSSpec.name[ 0 ] + 1 );

fileError = FindFolder(
              kOnSystemDisk,
              kAppleMenuFolderType,
              kDontCreateFolder,
              &aliasSpec.vRefNum,
              &aliasSpec.parID );

aliasSpec.parID = theFolderID;

// Now call FSpCreateResFile(), etc.
```

Listing 6. Settng up to create the new alias.

Logging to a file

Logging data to a file is easy, as long as you're appending. This next example appends to a text file. The record format looks like this:
Date <tab> Time <tab> Path to file

For example:

```
Wed, May 12, 1999  10:47:05 PM  Devt::tmp data
```

Since the user may delete the log file at any time, ensure it exists before opening.

```
  FSSpecPtr  logFSSpecPtr;
  OSErr      fileError;

  fileError = HCreate(
    ( *logFSSpecPtr ).vRefNum,
    ( *logFSSpecPtr ).parID,
    ( *logFSSpecPtr ).name,
    kLogFileCreatorType,
    kLogFileType );
```

Open, and move to the end of, the file.

```
  short  fileRef;

  fileError = FSpOpenDF( logFSSpecPtr,
    kSharedPermission, &fileRef );

  SetFPos( fileRef, fsFromLEOF, 0 );
```

Using the current time as a timestamp avoids the setup associated with (yet another call to) `PBGetCatInfo()`. But it's less accurate. As long as we get called soon enough, the difference may only be a few seconds.

```
  unsigned  long  theTime;
```

```
GetDateTime( &theTime );
```

Convert the timestamp into a human-readable date. Then write it out.

```
Str32  theString,

IUDateString( theTime, abbrevDate,
              theString );

dataCount = theString[ 0 ];
```

Move the date string into a buffer, then write the buffer to the file. Remember to initialize any pointers (not shown).

```
long   dataCount;
short  i;
Ptr    dataPtr;

// Faster than BlockMove()?
for ( i = 0; i < dataCount; i++ )
  dataPtr[ i ] = theString[ i + 1 ];

fileError = FSWrite( fileRef,
            &dataCount, dataPtr );
```

Adding a <tab> between fields improves readibility.

```
*dataPtr = '\t';
dataCount = 1;

fileError = FSWrite( fileRef,
            &dataCount, dataPtr );
```

Adding the time string is done the same way, but is not shown.

Next, get the volume name.

```
ParamBlockRec  theVolume;
OSErr    fileError;

theVolume.volumeParam.ioCompletion
  = 0L;
theVolume.volumeParam.ioNamePtr
  = ( StringPtr )&theString;
theVolume.volumeParam.ioVRefNum
  = theVRefNum;
theVolume.volumeParam.ioVolIndex
  = 0;

fileError = PBGetVInfo( &theVolume,
                        true );
```

Writing the volume name to the file is done the same way as the date string. Don't forget to add the two colons after the name.

Next, iterate over the directories in the file's path.

```
CInfoPBPtr  thePBPtr;

fileError = noErr;

while ( fileError == noErr ) {

  thePBPtr->dirInfo.ioCompletion =
    0L;
  thePBPtr->dirInfo.ioNamePtr =
    tempStringPtr;
  thePBPtr->dirInfo.ioVRefNum =
    theVRefNum;
  thePBPtr->dirInfo.ioDrDirID =
    tempFolderID;
  thePBPtr->dirInfo.ioFDirIndex = -1;

  fileError = PBGetCatInfo( thePBPtr,
                            true );

  while ( thePBPtr->dirInfo.ioResult
          == 1 ) {
    ;
  }
```

As long as the result refers to a directory, use the id to find the parent of this directory.

```
  if ( ( thePBPtr->dirInfo.ioResult
         == noErr ) &&
       ( fileError == noErr ) &&
       ( thePBPtr->dirInfo.ioFlAttrib
         & 0x0010 ) )
    thePBPtr->dirInfo.ioDrDirID =
      thePBPtr->dirInfo.ioDrParID;
}
```

Each iteration will place the name of the directory in the dirInfo.ioNamePtr field. This is the value we want to print, followed by a colon. However, since we're iterating up the directory tree, we should store the names temporarily, then write them out in the correct sequence after reaching the root.

Finally, write the name of the file, followed by a <return>.

```
  StringPtr fileNamePtr;

  dataCount = fileNamePtr[ 0 ];

  for ( i = 0; i < dataCount; i++ )
    dataPtr[ i ] =
      fileNamePtr[ i + 1 ];

  dataPtr[ i ] = '\r';
  dataCount++;

  fileError = FSWrite( fileRef,
    &dataCount, dataPtr );
```

Dispose of any pointers, close the file, and flush the volume.

```
  DisposePtr( dataPtr );

  FSClose( fileRef );

  fileError = FlushVol( NIL,
    ( *logFSSpecPtr ).vRefNum );
```

Listing 7. Logging new file info.

Now let's look at notification from a developer's perspective.

Callback functions

Callbacks are great from the perspective of time. But, they can be difficult to setup initially. And you need to make sure you don't call a routine that has moved! This will certainly cause problems if the application you're calling has quit.

Callback registration

Registering a callback requires that both sides know what to expect. For example, we can use one registration mechanism if we can distinguish between the types of activities that should be sent to a particular callback.

```
enum {
  kCreateFlag,
  kDeleteFlag,
  kCopyFlag,
  kRenameFlag
};
```

Similarly, the structure of the data being exchanged must be defined. Here, we've added a few fields to those defined by an FSSpec. This helps the caller in several ways:

- one callback may be used for multiple traps
- the sender can populate the structure with the most commonly used fields

The caller still has access to the various File Manager calls to get more information if necessary.

```
typedef  struct  FWData {
  short  theTrapId;
  short  theVRefNum;
  long   theParID;
  Str63  theString;
  OSType theFileType;
} FWData, *FWDataPtr;
```

This structure defines the data sent during an actual registration. The function address is sent, and the recipient needs to create a `UniversalProcPtr` based on the action specified (see the previous `enum` definition).

```
typedef  struct  FWSubscribe {
  short  theAction;
  long   *theCallbackAddr;
} FWSubscribe, *FWSubscribePtr;
```

The callback convention looks as follows:

```
ProcInfoType uppCreateFileProcInfo =
  kPascalStackBased
  | RESULT_SIZE( kNoByteCode )
  | STACK_ROUTINE_PARAMETER( 1,
      SIZE_CODE( sizeof( FWData ) ) )
  ;
```

Sending in a registration request might look something like this:

```
OSErr  theErr = noErr;
SelectorFunctionUPP  myGestaltUPP;
FWSubscribePtr  theFWSubscribePtr;
```

Locate the function handling callback registration. Using `Gestalt()`, you need to know the advertised signature.

```
theErr = ::Gestalt( kTargetSignature,
  ( long * )&myGestaltUPP );

if ( theErr == noErr ) {
```

Allocate and fill a structure specifying the type of activity we're interested in, and the address of the callback function.

```
  theFWSubscribePtr =
    ( FWSubscribePtr )
    NewPtr( sizeof( FWSubscribe ) );

  if ( theFWSubscribePtr != nil ) {

    theFWSubscribePtr->theAction =
      kCreateFlag;

    theFWSubscribePtr->theCallbackAddr
      = ( long * )&MyCreateCallback;

    theErr = CallSelectorFunctionProc(
      myGestaltUPP,
      GESTALT_ADD_CALLBACK,
      ( long * )theFWSubscribePtr );

    DisposePtr(
      ( Ptr )theFWSubscribePtr );
  }
}
```

Listing 8. Registering a callback: requester.

On the receiving side, the corresponding registration process looks like this in the `Gestalt()` handler.

```
  theAction = ( ( FWSubscribePtr )
              theResponse )->theAction;

  if ( theAction == kCreateFlag )
    gCallbackProcPtr =
      NewRoutineDescriptor(
        ( ProcPtr )( ( FWSubscribePtr )
        theResponse )->theCallbackAddr,
        uppCreateFileProcInfo,
        kPowerPCISA );
```

Listing 9. Registering a callback: receiver.

We distinguish between the activities in case we want to manage `_Create` callbacks differently. Notice that a UPP is built here. This example also assumes the PowerPC instruction set architecture.

This example uses a single global variable to hold the callback UPP. Using a list or queue would increase the potential number of clients. [Rentzsch99] addresses this issue; you should definitely check it out.

Callback execution

Call the callback via `CallUniversalProc()`. Don't forget to populate the data structure first.

```
FWData    theFWData;

theFWData.theVRefNum =
  gParamBlockCopy->fileParam.ioVRefNum;
// etc.

CallUniversalProc( gCallbackProcPtr,
   uppCreateFileProcInfo, theFWData );
```

Listing 10. Calling a developer-provided routine with the new file info.

On the receiving side, the stack needs adjusting to account for the first two params to `CallUniversalProc()`.

Upon entering the callback, increment the stack pointer by two words:

```
asm void  PrologGlue() {
  addi  sp, sp, 0x08
  blr
}
```

When leaving the callback, decrement the stack pointer by the same amount:

```
asm void  EpilogGlue() {
  subi  sp, sp, 0x08
  blr
}
```

This particular callback is part of a PowerPlant project, and simply relays the incoming data to a separate method in the application:

```
pascal  void    MyCreateCallback(
                  FWData theFWData ) {
  PrologGlue();
```

```
  ( ( FileWatcherDemo* )
    parentApp )->HandleCreate(
                  theFWData );

  EpilogGlue();
}
```

Listing 11. Inside the callback.

Figure 3 shows the raw information being fed (via the callback just discussed) to a text area for diagnostic purposes. It displays the FSSpec-related fields associated with a file creation.

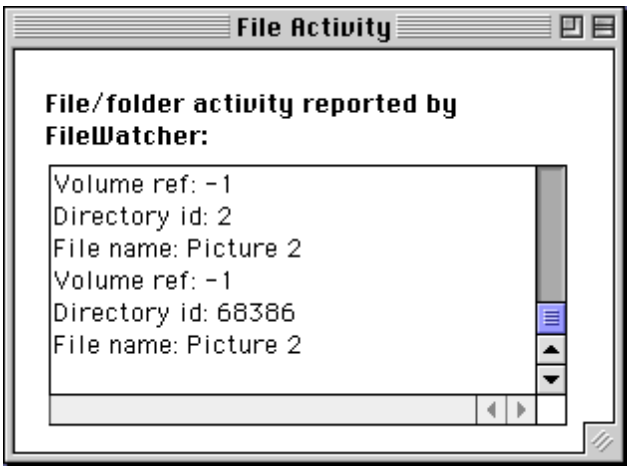

Figure 3. Demo app receiving info from the file system watcher.

Sending Apple Events

Apple Events provide another convenient mechanism to transmit data from one component to another. This section discusses the sending of file creation information from the patch to the background app.

In order to support scripting, and also to allow for checking of required data elements, the background app contains an `'aete'` resource that defines a custom Apple Event for handling file creation. This example uses an FSSpec, rather than the FWData type discussed in the section on callbacks. The FSSpec is sent as the direct parameter for this event.

```
resource 'aete' ( 0 ) {
  0x1, 0x0, english, roman, {
    "File Watcher Suite",
    "Specialized stuff.", 'FWAE', 1, 1,
    {
      //  Events
      "handle new file",
      "A new file to record.",
      'asd9', 'newf',
      noReply, "No reply.",
      replyOptional, singleItem,
      notEnumerated,
      reserved, reserved, reserved,
      reserved, reserved, reserved,
      reserved, reserved, reserved,
      reserved, reserved, reserved,
      reserved,
      '****', "FSSpec for new file",
      directParamRequired, singleItem,
      notEnumerated,
      doesntChangeState,
      reserved, reserved, reserved,
      reserved, reserved, reserved,
      reserved, reserved, reserved,
      reserved, reserved, reserved,
      {}
    }, {}, {}, {},
  },
};
```

Listing 12. Resource defining a file create Apple Event.

Preparing and sending an Apple Event is addressed in [Little91]. This code, located in the patch, populates an event with the most recent `_Create` data:

```
AEAddressDesc  theAddressDesc;
AppleEvent  theAppleEvent;
FSSpec  theSpec;
OSErr  theError;

theError = FSMakeFSSpec(
  gParamBlockCopy->fileParam.ioVRefNum,
  gParamBlockCopy->fileParam.ioDirID,
  gParamBlockCopy->fileParam.ioNamePtr,
  &theSpec );

theError = AECreateAppleEvent(
  'asd9', 'newf', &theAddressDesc,
  kAutoGenerateReturnID,
  kAnyTransactionID,
  &theAppleEvent );

theError = AEPutParamPtr(
  &theAppleEvent, keyDirectObject,
  typeFSS, &theSpec,
  sizeof( FSSpec ) );
```

Listing 13. Populating an Apple Event.

The receiver of this Apple Event must be prepared to handle it. Here, the background app registers the function `HandleCreate()` as the recipient of file creation events. The class and id match those found in the `'aete'` resource.

```
const OSType  kTestEventClass ='asd9';
const OSType  kCreateEventId = 'newf';

AEInstallEventHandler(
  kTestEventClass, kCreateEventId,
  (AEEventHandlerProcPtr)HandleCreate,
  0, false );
}
```

The function `HandleCreate()` will extract the data from the event, and pass it off for additional processing. For our custom event, the `itemsInList` value should always be one.

```
FSSpec  theFSS;
OSErr   theErr;
SFTypeList  theTypeList;
Size    actualSize;

for ( i = 1; i <= itemsInList; i++ ) {
  theErr = AEGetNthPtr( &docList, i,
      typeFSS, &theKeyword, &typeCode,
      ( Ptr )&theFSS, sizeof( FSSpec ),
      &actualSize );

  theErr = DoMakeAlias( theFSS );
}
```

Listing 14. Handling an Apple Event.

Conclusion

The Mac OS file system can provide a lot of useful information regarding file and folder activities. The difficulty lies in determining what information you desire, and obtaining that information in a timely manner (i.e. as it happens). Once you have the raw data associated with a file system event, translating it into something useful, and then communicating that info to a user or developer is straightforward. This paper presented several ways of doing all of the above. Hopefully, you can develop new products or enhance existing products using the techniques described here.

Bibliography

[Apple93] Apple Computer, Inc. AppleScript Scripting Additions Guide. Addison-Wesley Publishing Company, Reading, MA. 1993.

[Apple94] Apple Computer, Inc. Inside Macintosh: Operating System Utilities. Addison-Wesley Publishing Company, Reading, MA. 1994.

[Apple96] Apple Computer, Inc. Technote 1070: Background-Only Applications. October, 1996.

[Little91] Little, Gary and Swihart, Tim. Programming for System 7. Addison-Wesley Publishing Company, Reading, MA. 1991.

[Mark94] Mark, Dave. Ultimate Mac Programming. IDG Books Worldwide, Foster City, CA. 1994.

[Rentzsch99] Rentzsch, Jonathan. Atomicity: Concurrent Data Access Without Blowing Up. Red Shed Software, Schaumburg, IL. 1999.

[Thompson94] Thompson, Tom. Power Macintosh Programming Starter Kit. Hayden Books, Indianapolis, IN. 1994.

[Zobkiw95] Zobkiw, Joe. A Fragment of Your Imagination. Addison-Wesley Publishing Company, Reading, MA. 1995.